# Scientific Computing with Dependent Types for Reliability

SciDeR (version: 2018-04-10 09:04)

## 1 Purpose and aims

Dependently-typed programming systems, such as Agda [20], Idris [5], and Coq [25], are among the most important recent advances in computer science.[1] They provide an environment in which the programmer can both formulate the requirements that the code must satisfy and implement the code. Moreover, the implementation can be guided by the requirements, and parts of it can be automatically "filled in" by the system. The environment then checks that the implementation really does satisfy the requirements. Thus, the result is a provably correct program, with a machine-checked proof. At the moment, this is the highest standard of correctness that we have.

Contrast this method of "correct by construction" programming with the standard practice in today's software industry (and academia). In standard practice, the requirements are usually formulated in natural language, requiring an effort of interpretation from the programmer. Often, the results of this effort are not made explicit, making it difficult to see whether they are the intended ones or not. Based on their interpretation, the programmers produce an implementation. Finally, this implementation is **tested**. The invention of adequate tests is somewhat of an art, and their correctness and completeness can be difficult to ensure [12].

That dependently-typed programming environments are now mature enough to improve the current situation can be seen by their success in, e.g., producing a verified C compiler, CompCert [15]; developing database access libraries which statically guarantee that queries are consistent with the schema of the underlying database [21]; implementing secure distributed programming [24]; implementing resource-safe programs [18, 6]; and many others.

However, there are so far almost no dependently-typed libraries for **scientific computing**, **machine learning** or **artificial intelligence** algorithms such as **deep learning**, **clustering**, **self-organising maps**, etc. What characterises these applications is that they model aspects of the "real" world, not just the world of digital systems (as in the above examples). Such applications comprise the bulk of software used in science and engineering. Typical examples are: models of the climate, the economy, the energy system, public health, weather forecasting, autonomous systems, and so on. They are used on a daily basis in industry, decision making, health care, traffic control, etc. It is clear that ensuring their correctness would be greatly desirable.

The only substantial example of dependently-typed programming in the context of scientific computing is **IdrisLibs**, a collection of application-independent components developed at the Potsdam Institute for Climate Impact Research together with Chalmers in order to solve sequential decision problems arising in climate impact policy analysis and advice [2].

We aim to use the results and the experience gained in the course of this activity in order to develop a **software infrastructure for reliable scientific computing and machine learning**. The infrastructure will consist of a collection of **open source software components** in the dependently-typed programming language Idris, **methods** for developing and maintaining large dependently-typed libraries, and **tools and techniques** for integrating existing high-quality libraries in a dependently-typed environment. This infrastructure should contribute to the creation of a community of users, both from industry and from the academic world, producing reliable

[1] It is perhaps interesting to note that this is mostly a European, and, in particular, a Swedish achievement. All the main environments for dependently-typed programming are developed in Europe (Agda in Sweden, Idris in the UK, Coq in France) and they are all based on the intuitionistic type theory developed by the Swedish mathematician and philosopher Per Martin-Löf [16].

applications in engineering and scientific fields such as autonomous vehicles, medical research, climate impact research, modelling financial markets, Big Data analysis, etc.

The availability of such a software stack will not just actively promote **improved software applications**, but will also enable **new fundamental research**, especially in fields where **computer models** are used to replace **laboratory experiments**. These fields include the social sciences, climate research, parts of medical research, etc., where such experiments are not possible either for practical or ethical considerations. Such computer models often embody theories that are only partially specified (and of which the scientists themselves are not necessarily aware). By making the assumptions in the code explicit in the form of properties, high-level types, and postulates, a dependently-typed programming language makes it possible to formulate **new theories** about the domain of the code. For an example, see the formalisation of the notion of **vulnerability** in the context of climate change [14].

What we call "software" or "program" in this proposal is not just "a sequence of instructions, written to perform a specified task with a computer" [Wikipedia, Computer Program]. In **SciDeR** terminology, a "program" is both a high-level description of a problem or task and a proven correct procedure for solving that task. Thus, a library for a specific application domain will be viewed, on one hand, as a collection of building-blocks for describing problems in that domain, and on the other hand as a toolkit of methods for computing within that domain. From this perspective, the iterative process of devising a "program" to match a "specification" becomes a way of exploring and understanding a domain from a computational point of view. We rely on languages with dependent types to express specifications, implementations and proofs in a common framework.

## 2 State-of-the-art

As mentioned in the introduction, there are, with the exception of IdrisLibs, no substantial examples of libraries for scientific computing and machine learning in the main dependently-typed programming environments. The closest-related efforts so far have concentrated on using these environments in order to formalise mathematical theories relevant for computing (this is especially the case in Coq, e.g. [7]), or in computations with exact real numbers [17]. Neither of these developments provides access to existing high-quality libraries, nor have we found attempts to formulate the properties of such library functions. Isolated implementations of linear algebra methods [9] or neural networks [8] are of limited usability, in terms of efficiency and features.

The project that most closely resembles our intentions is SciPy, which represents, according to its homepage, several related but distinct entities:

- the SciPy ecosystem, a stack of open source software for scientific computing in Python,
- the community of people who use and develop this stack,
- some conferences dedicated to scientific computing in Python (SciPy, EuroSciPy, SciPy.in),
- the SciPy library, one component of the SciPy stack, providing many numerical routines.

SciPy provides access to excellent numerical libraries in order to enable the users to glue them together using the Python programming language for rapid prototyping. In a sense, Python is the ideal vehicle for such an environment, due to the ease with which external programs can be integrated. However, this is achieved by removing even the limited guarantees that the libraries come with. In the square root example, the Python interface gives a function for which even type errors are only recognised at run-time, increasing the difficulty of building reliable applications.

We conclude this section with two examples of how **SciDeR** can foster new research in the area of dependently-typed programming languages.

In critical applications, recognising type errors at run-time is not enough. In these cases, the developers will be required to provide evidence for the correctness of the postulates. One of

the ways in which this can be achieved, besides re-implementing the code in Idris, is by using tools for **program verification**, such as K[22]. Program verification is a complementary method to dependently-typed programming, allowing the (in general more difficult and expensive) a-posteriori proof of correctness, instead of building the code in a provably correct way from the very beginning. One of the research directions we hope to trigger is that of integrating the results of such program verification in dependently-typed programming environments.

In section 5.1, problem 1 explains how the entanglement between proofs and algorithms can simplify the development effort, while, at the same time, it can lead to brittle code that is very difficult to maintain and change. There is a possibility to carry over at least some of the advantages of the "tangled" approach in the context of incremental development we argue for here. This requires language support for **reflection**: the ability to access the program text as a data structure within the program itself. While Idris has some support for reflection, both the technology and the "best practices" are in need of further research. This is one direction in which our project might "nudge" the developers of dependently-typed programming languages.

# 3 Significance and scientific novelty

The proposed project **SciDeR** aims to provide a **novel instrument** in the form of a **software stack for reliable scientific computing and machine learning**, new **methods** for developing and maintaining large dependently-typed libraries, and **tools and techniques** for integrating existing high-quality libraries into a dependently-typed environment. Through this infrastructure, we hope to contribute to the industry-science dialogue by the creation of a community of users from both sides. Given the ubiquity of scientific computing and machine learning in both industry and the academic world, we believe that we are **targeting an area of current and future growth** for both the private and the public sector.

Finally, the availability of what is, in essence, an instrument for mathematical formalisation will lead to **new and exciting fundamental research**, especially (but not exclusively) in the social sciences. By forcing the scientists to express their assumptions explicitly and formally, dependently-typed programming languages can aid them in developing mathematical theories about the domain of the code. As examples, see the formalisation of the notion of "vulnerability" in the context of climate change [14], and the notion of "avoidability" in policy advice [2].

We hope to foster new research in dependently-typed programming languages, making it easier to use the "increasingly correct computing" methodology [11].

> The road to wisdom?
> — Well, it's plain
> and simple to express:
> Err
> and err
> and err again,
> but less
> and less
> and less.
> — Piet Hein

# 4 Preliminary and previous results

Early work in the direction of SciDeR was published by the PI already in 2009 (Dependent types for relation program derivation, [19]). This lead to the related VR project "Strongly Typed Libraries for Programs and Proofs" (StrongLib, 2011–2014). As part of StrongLib we wrote a "proof-of-concept" paper on "Dependently-typed programming in scientific computing" [11]. That paper can be said to be the starting point of the SciDeR project. At around the same time, we also published a paper describing cases where proving is actually more effective than testing, also here in the context of dependently typed programming, [12].

After StrongLib we worked on a mathematical theory for, and practical implementation of, parallel parsing [1]. It turns out that the dependently typed formalisation of parsing is also more generally useful as a correct-by-construction implementation of block matrices [23].

In parallel, Botta, Jansson and Ionescu initiated a more ambitious case study: formalising sequential decision problems and their solutions (using Bellman's backward induction) for use in policy advice. This lead to three joint journal papers so far: in Logical Methods of Computer Science (LMCS), in the Journal of Functional Programming (JFP) and in Earth System Dynamics (ESD). The first paper describes the theory and Idris implementation of generic, monadic, Sequential Decision Problems [4]. The following journal paper is a extension of the previous paper on SeqDecProbs in the direction of policy advice and avoidability. The underlying Idris implementation was refactored and extended to handle the new application [2]. Finally, the third journal paper is a very recent (2018) application of the results of the two previous papers to the problem of optimal emission policies in the context of climate impact research [3].

The application-independent components that were needed to implement the framework for sequential decision problems form the basis of IdrisLibs. The experience gained in the development of IdrisLibs has led to the realisation of the difficulties involved in using dependently-typed programming environments in the context of scientific computing, and of the most promising ways in which to tackle them. In a sense, IdrisLibs represents the best argument both for the feasibility, and for the necessity of the research programme outlined here.

Last but not least, the applicants worked on developing a BSc level course on "Domain Specific Languages of Mathematics" which has now been taught three times at Chalmers [13]. The development of this course, and the associated course material, has helped prepare the SciDeR project. The course itself will serve as a source of talented students for BSc, MSc and PhD thesis work on our software infrastructure for reliable scientific computing.

# 5 Project description

In order to understand the problems that arise when attempting to build such an infrastructure, it is important to have an elementary understanding of dependently-typed programming. Perhaps the simplest example is that of a function that computes the square root of its argument. Virtually all mainstream languages provide a square root function that accepts as argument a floating point number. Most compilers will prevent the programmer from passing a list or a character to the square root function, which would constitute a **type error**, but they will **not** prevent passing a negative number, which will generally result in a run-time crash of the system (this kind of error is responsible for many of the famous software systems failures, such as the destruction of Ariane 5 in 1996). In a dependently-typed programming language, by contrast, one can define the type of **non-negative number**, and the square root function will only accept arguments of this type. The type checker (part of the compiler) can now also flag as an error a programmer's attempt to pass a negative value to the square root function, and no such run-time errors are possible.

The types that can be formulated in such languages are not limited to "low-level" data (like numbers). For example, we can formulate types for "resource-safe operation", "privacy-ensuring protocol", but also for "avoidable state" [2] or even "measure of vulnerability to climate change" [10]. For such high-level types, establishing the correctness of a program via testing is often not a realistic option [12].

Note that, even in the simple example of the square root, the analysis of the types can be further carried out. One can generalise the input type, from floating-point numbers to more inclusive notions, but one can also formulate the conditions fulfilled by the output of the square root algorithm: squaring the output should, within limits imposed by the precision of operations, give back the argument. This kind of analysis leads to a **mathematical formalisation** of the computational content, and can form the basis of **new theories** underlying computational models. This can be very useful in areas where theories are lacking, such as policy analysis, cf. [2, 3].

## 5.1 Major problems

With this understanding, we can now explain the most important obstacles in creating a software infrastructure for reliable scientific computing and machine learning.

1. **Difficulties in maintaining large dependently-typed libraries**. Dependently typed libraries are collections of **data structures** and **algorithms**, and of their machine checkable **properties**. It is often the case that the properties of an algorithm are formulated in a way that crucially depends on the **implementation**, in order to simplify the proofs of correctness. However, if the implementation is modified, for example in order to make it more efficient, all corresponding properties have to be reformulated (and all proofs of correctness need to be redone). Because of this, collections of properties that depend on specific implementations are essentially non-maintainable.

   Similarly, the richness of dependently-typed programming languages allows the creation of **data structures with properties**. For example, where standard programming languages are only able to offer a list data structure, a dependently-typed one gives the programmers datatypes for sorted list, lists of non-zero elements, lists of a given length, lists of permutations, etc. In turn, this can lead to a proliferation of functions operating on these data structures. For example, a sorted list is not just a list, but a list with additional information certifying to its being sorted, and so cannot be passed as an argument to a library function such as `length`, which expects a plain list. There are known ways to cope with these difficulties, but they require considerable effort and discipline on the part of the programmers.

2. **Difficulties in integrating existing libraries**. A huge amount of effort has been spent on high-quality libraries for scientific computing, such as BLAS, LAPACK, NAG, and many others. Usually, the errors in scientific computing applications are not due to the library functions themselves, but to their (mis-)use by the users. In the example of the square root function, the quality, efficiency or precision of the square root algorithm is not the problem: the error comes from making an unintended use of it.

   Integrating high-quality existing libraries is an important part of creating an infrastructure for reliable scientific computing, but it also one of the most challenging tasks. The main problem is that, by virtue of their being external to the dependently-typed programming environment, the functions provided by these libraries are “black boxes” and, as such, their properties cannot be proved within the system. For example, we can be certain that the square root function provided by an external library really does deliver the requisite approximation to the square root of its (non-negative) argument, but we cannot have a machine-checked proof of this fact.

3. **Lack of precise specifications for scientific computing.** In contrast to the libraries mentioned above, algorithms for scientific computing come with an additional problem: it is very difficult to pinpoint the properties that these algorithms have. For example, when using a genetic algorithm to maximise a fitness function, it is very difficult to be precise about the quality of the solution. In general, it will not be optimal, but how far it is from the optimum is something that the current theory cannot tell us. Similarly, we can expect a deep learning network to interpolate the training set and to extrapolate beyond it, but how good the interpolation or extrapolation is cannot currently be determined a-priori.

   At present, whether a machine learning algorithm will be effective on a given problem is largely a matter of experimentation, of trial and error. That means that integrating them will require a different kind of postulate than the more traditional and constrained algorithms: namely, via a **classification and formalisation** of the kinds of problem that the various algorithms are useful for. This will at least guide the users in the selection of the most appropriate algorithm for their tasks. It is important to ensure that this formalisation is **open-ended**, and that newer theoretical results can be incorporated as they appear.

## 5.2 Theory and methodology

The methodology used to address these problems is that of taking an **incremental** approach towards **software correctness**, that we call **increasingly correct computing** [11].

We highlight three distinct features of this approach:

1. **Separation between specification and implementation**

   A crucial element is the separation between specification and implementation. This allows us to address problem 1 described above, that of building maintainable dependently-typed libraries, by disentangling the algorithms and data structures from their properties.

   This separation of concerns has a number of interesting consequences: first, it allows us a **faster development of computational content**, since the proofs of correctness can be "filled in" after the implementation.

   In our square root function example, we would be able to implement a square root algorithm, declaring its return type to be simply floating-point. The property that the returned value, when squared, is equal to the argument to the function (up to numerical precision), will be stated and proved separately. Note that the input type to the square root function must still be "non-negative floating-point value" (or similar). Otherwise, the type checker could not prevent the programmer from passing a negative value, thus generating a run-time error. In other words, the separation proposed here does not mean that functions can keep the types they would have in non-dependently typed languages.

   Second, since properties are formulated independently from the implementation, we can now build **libraries of properties**. In this way, it becomes possible to organise community efforts in a clear and unambiguous way by publishing collections of **specifications** to be implemented in an efficient manner.

2. **Conditional correctness**

   The integration of existing libraries into the dependently-typed environments is generally accompanied by a lowering of the standard of correctness described above. Since to the type-checker a foreign function is effectively a black box, the properties attributed to it cannot be verified: we cannot have machine-checkable proofs of correctness. The only way to make use of knowledge about foreign functions is to enter it into the environment in the form of a **postulate** or **axiom**: a property that is accepted by the environment on the "authority" of the expert. Once entered into the system, these postulates represent a logical interface to the foreign functions that allow us to use them in a provably reliable way. Provided, of course, that the postulated properties really hold!

   Conditional correctness (correctness up to a collection of postulates) is our method of choice for solving the difficulty of integrating existing libraries, problem 2 above. The selection and formulation of properties to be postulated is a non-trivial task. On the one hand, given that an erroneous postulate can endanger the whole system, one is tempted to postulate only weak properties. In the example of the external square root function, this would correspond to postulating only that the result will be of type floating-point number. While accurate, this is too weak to be useful in reasoning about applications that then use the square root function. The opposite temptation is to make the postulate very strong, in order to simplify the reasoning. This would correspond to postulating that the result of the square root function is the **exact** square root of its argument, omitting the limited precision of floating-point numbers.

   Striking the right balance between (too) weak and (too) strong specifications will be one of the main research challenges in SciDeR.

3. **Incremental development**

   The ideal of machine-checkable proof of correctness may sometimes be extremely difficult to achieve, even in cases in which the computational content is entirely implemented in the dependently-typed environment. This is often the case in proving properties of algorithms for scientific computing and machine learning, which rely on theorems from mathematical analysis. In such cases, the programmer might treat the available mathematical knowledge as the "foreign functions" above, and simply postulate the properties. Again, this represents a lowering of the maximal standard of correctness, in the interest of reasonable development time.

   We can think of conditional correctness as the minimum level that one can accept in a dependently-typed system. Not all the properties have a (machine-checkable) proof, but they are all explicit, and, if satisfied, guarantee the correctness of the application.

   Once conditional correctness is established, we can, when necessary, begin to **raise the correctness level by incrementally replacing postulates** with actual machine-checkable proofs. For scientific computing libraries one will accept that, in many cases, machine-checkable correctness cannot be achieved. This is particularly true for libraries that rely on efficient floating point approximations of real numbers. Conditional correctness is likely the best that we can realistically think of achieving for software to assist decision making under uncertainty, pattern recognition and, by and large, engineering applications.

   Incremental development (increasingly correct computing) is our proposed cure for problem 3 above, the lack of precise specifications for scientific computing.

The methodology of increasingly correct computing allows us to accelerate the development time at the expense of weakening the standard of correctness, from full proof of correctness to conditional correctness. This is a pragmatic choice, that allows the **rapid integration of high-quality external libraries**, where the highest standard of correctness would require their reimplementation. Having access to such external libraries means that one can immediately develop applications, rather than being bogged down in infrastructure. Therefore, the chances of attracting users from both industry and the academic world will be correspondingly higher.

Spearheading a community effort is a key part of our methodology. The development of IdrisLibs would not have been possible without a close collaboration with the developers of the language. With **SciDeR**, we will strengthen this collaboration.

In order to ensure the contribution of the community, we will, from the beginning of the project, publish specifications to be taken up by implementors on a popular platform for collaborative development like GitHub or GitLab, together with examples of using foreign function interfaces to popular libraries. At the same time, we will present the project results to the community through contributions to high-level conferences and the standard social media tools (blogs, Twitter, etc.). We will also organise a summer-school with training sessions for external students and users.

In this enterprise, Chalmers will play an essential role at the interface between research, education and scientific applications. We will exploit existing collaborations with research and industry through the LiVe4CS consortium[2] in order to train the next generation of highly skilled, entrepreneurial researchers and apply novel, reliable scientific computing and machine learning libraries for sustainable solutions.

[2]Lightweight Verification for Complex Systems (LiVe4CS) is a European consortium for PhD education in Computer Science with academic sites (HWU, USTAN, USTR, UGOT, LMU, UU), large companies (Intel, Microsoft, Mozilla, Siemens, JetBrains), SMEs/Start-ups/R&D labs (BTL, ESL, Quviq, WT, SCCH, Stat-Up), and specialised research centres (MPI, PIK, ECR).

## 5.3 Time plan and implementation

We aim to make the results of the project available as soon as possible, in order to maximise the chances of creating a community of users to apply, improve, and extend them. The milestones are:

1. End of month 9: Collections of **specifications** and **verified properties** for the most basic Idris libraries: Prelude, Data, etc.
2. End of month 18: **Idris binding to essential external numerical libraries**, together with a **tools and techniques for integrating external libraries in a dependently-typed ecosystem**. This is an essential goal of the project, and we aim to have it completed early.
3. End of month 24: Creation of example applications using the new functionality, in particular of **reliable infinite horizon decision problems**. These will help improvements in tools and techniques created in the first project year, and attract new users, as such problems are frequently encountered in game theory, economics and financial models. Applications are also essential for testing the usability and maintainability of the software stack.
4. End of month 36: **Idris bindings to a machine learning library**, such as TensorFlow. The integration of machine learning libraries poses new challenges, due to the lack of formal understanding of most of the algorithms. Dealing with the associated imprecision is an important test of the tools and techniques for external libraries.
5. End of month 48 (end of the project): Creation of one or more **reliable reinforcement learning** application, such as AlphaZero, and release of a **method for the development and maintaining** of a dependently-typed infrastructure. By this time, we hope to see the first-year results used in industry and the academic world. Their existence would indicate a successful run of the project.

## 5.4 Project organisation

The present research and development group consists of the following three participants, whose collaboration is more than a decade old.

1. Main applicant: **Prof. Patrik Jansson** is Full Professor of Computer Science and Deputy Head of the department of Computer Science and Engineering at Chalmers. His research focuses on systems for constructing correct and reusable software with the goal of developing the programming languages of the future. He is actively promoting technology transfer through open source software at GitHub. He has published 43 papers (h-index 20), has been PI on 3 EU projects, has successfully supervised 4 PhD students and is currently examiner of 3.
2. Co-applicant: **Dr. Cezar Ionescu** is Associate Professor of Data Science at the Oxford University Department for Continuing Education, and guest researcher at Chalmers. His main interests include functional programming with dependent types, program calculi, domain-specific languages, and the role of computing science in education. His recent work focuses on correctness of scientific computing and machine learning algorithms.
3. External collaborator: **Dr. Nicola Botta** is senior scientist at Potsdam Institute for Climate Impact Research. He has been working on sequential decision problems in the context of greenhouse gas emissions, on agent based models of exchange economies and on numerical methods for partial differential equations.

In addition, the project resources will allow us to extend the group with a PhD student:

- For the first two years, her main responsibilities will be the development of theories, tools and techniques for integrating functionality from existing numerical libraries into a dependently typed software stack. This will be done in collaboration with academic users of the stack with the aim of preparing a licentiate thesis on “Increasingly correct scientific computing”.
- For the last two years, her main contributions will be in the development of applications using the software stack, identifying and solving important research problems, contributing to, and documenting, the resulting methodology for large dependently-typed programs and proofs. This will be done in collaboration with external users of the stack to help technology transfer in one direction and to provide a steady stream of relevant problems to serve as challenging test cases.

After the SciDeR project is done, her fifth and final year will be mainly devoted to writing up the PhD thesis on “Increasingly correct scientific computing”.

### 5.5 Local host

The Department of Computer Science and Engineering (CSE) is strongly international, with about 70 faculty and 100 PhD students from about 30 countries. CSE has a leading role in the fields of functional programming, type theory, domain-specific languages, generic programming, machine learning, and more.

The division of Functional Programming has six professors and 20 other researchers and PhD students. The focus of the division is the application of programming technology, and functional programming techniques in particular, to solve practical problems. As a result, there is a strong emphasis in the division in writing programs that solve real world problems. In addition, there is a strong culture of valuing general and “elegant” solutions.

The size and focus of the division, as well as its history, make it unique in the world and it enjoys a very strong international reputation, as evident both in the number of highly cited publications, the number of memberships in premier conference program committees, as well as in the number of invited lectures given by members of the division. A strength of the division is the close contact it enjoys with industry, both through startup companies spun off from the division as well as through regular joint projects with industry or industrial research laboratories. These interactions provide a steady stream of relevant problems that both influence much of the long-term research as well as provide challenging and relevant test cases. The researchers in the division share to a very large extent a common vision for what constitutes a good research problem, what constitutes a good solution, and how a research result should be evaluated.

## 6 Other applications or grants

In March 2018 a supporting project application called “RISC” (with Jansson as PI) was handed in to the SSF ITM17 call “*Instrument, Technique, and Method Development Projects 2017*”. That project is more applied and would mainly fund two PostDocs, but the underlying theme of Increasingly Correct Scientific Computing is the same.

In January 2018 the LiVe4CS consortium handed in an application for a European Joint Doctorate (EJD) school on “Lightweight Verification for Complex Systems (LiVe4CS)” which has some overlap with SciDeR (one of 15 PhD projects is similar to SciDeR). Jansson is a co-applicant in LiVe4CS.

In March 2018, Jansson (as coordinator) handed in an application (EDGE - Exascale, Data, and Global Evolutions) for a large European HPC project with 19 partners and a budget of

around 80M SEK. The EDGE project would provide a very strong network for scaling up the SciDeR work to large scale computing resources.

## References


[1] J.-P. Bernardy and P. Jansson. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *Logical Methods in Computer Science*, 12, 2016.

[2] N. Botta, P. Jansson, and C. Ionescu. Contributions to a computational theory of policy advice and avoidability. *Journal of Functional Programming*, 27:1–52, 2017.

[3] N. Botta, P. Jansson, and C. Ionescu. The impact of uncertainty on optimal emission policies. *Earth System Dynamics Discussions*, 2018:1–24, 2018.

[4] N. Botta, P. Jansson, C. Ionescu, D. R. Christiansen, and E. Brady. Sequential decision problems, dependent types and generic solutions. *Logical Methods in Computer Science*, 13(1), 2017.

[5] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[6] E. Brady and K. Hammond. Resource-safe systems programming with embedded domain specific languages. In *Practical Aspects of Declarative Languages*, pages 242–257. Springer, 2012.

[7] G. Cano, C. Cohen, M. Dénès, A. Mörtberg, and V. Siles. Formalized linear algebra over elementary divisor rings in Coq. *Logical Methods in Computer Science*, 12(2), 2016.

[8] S. Chowdhury. A quick Idris implementation of @mstksg's "dependent Haskell" neural networks. Available from `https://gist.github.com/mrkgnao/a45059869590d59f05100f4120595623`.

[9] G. Gonthier. Point-free, set-free concrete linear algebra. In *Proceedings of the Second International Conference on Interactive Theorem Proving*. Springer, Berlin, Heidelberg, 2011.

[10] C. Ionescu. *Vulnerability modelling and monadic dynamical systems*. PhD thesis, Freie Universität Berlin, 2009.

[11] C. Ionescu and P. Jansson. Dependently-typed programming in scientific computing: Examples from economic modelling. In R. Hinze, editor, *24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, volume 8241 of *LNCS*, pages 140–156. Springer, 2013.

[12] C. Ionescu and P. Jansson. Testing versus proving in climate impact research. In *Proc. TYPES 2011*, volume 19 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 41–54, Dagstuhl, Germany, 2013.

[13] C. Ionescu and P. Jansson. Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. In *Proc. 4th Int. Workshop on Trends in Functional Programming in Education*, EPTCS. Open Publishing Association, 2015.

[14] C. Ionescu, R. J. T. Klein, J. Hinkel, K. S. Kavi Kumar, and R. Klein. Towards a formal framework of vulnerability to climate change. *Environmental Modelling and Assessment*, 14(1):1–16, 2009.

[15] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[16] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.

[17] C. Ming Chuang. *Extraction of Programs for Exact Real Number Computation Using Agda*. PhD thesis, Swansea University, 2011.

[18] J. Morgenstern and D. Licata. Security-typed programming within dependently-typed programming. In *International Conference on Functional Programming*. ACM, 2010.

[19] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009.

[20] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola, 2007.

[21] N. Oury and W. Swierstra. The power of Pi. In *Proc. of ICFP 2008*, pages 39–50. ACM, 2008.

[22] D. Park. Program verification in the K framework. Available from `https://github.com/kframework/k-legacy/wiki/Program-Verification`.

[23] A. Sandberg Eriksson and P. Jansson. An Agda formalisation of the transitive closure of block matrices (extended abstract). In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 60–61. ACM, 2016.

[24] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proc. of ICFP 2011*, pages 266–278, 2011.

[25] The Coq development team. The Coq proof assistant reference manual, 2009.